Project 2 Concurrency Report

Github

Program Structure:

The goal imath project is to create a program that takes in a P6 image and creates a new P6 image by applying the Laplacian filter to the original image. This program works by taking in the name of a file from the command line (This is done in main). Then using reading/parsing the header of the file, storing the height and width, and then parsing the RGB pixel data into an array of PPMPixel (A globally defined structure), all of this is done in the readImage function.

Then the apply_filter function will take the RGB data (that is stored in a PPMPixel array). Create THREAD (globally defined) threads, then each thread we will calculate its start point, and how much work it will do, and then run the threadfn function (We will discuss more of what this is below) which will take a parameter (A globally defined struct) that holds the original RGB pixel data, the new RGB pixel data (with the filter applied), width, height, start point, and how much work it will do. Each thread will do an even amount of work (Excluding the last thread which will do any additional work needed to finish, if the work can't be divided up evenly). The apply_filter function will also compute the time it takes to apply the filter using all threads and this time will be outputted in main.

The threadfn function will apply a Laplacian filter (of a size defined globally) to the original RGB pixel data. It will do this by multiplying the value of the filter with its corresponding pixel for each of its RGB color components. It will also make sure using the inRange function that the newly creates color components are in the range of possible RGB values (between 0 and 255), and if they are not in range sets them to the minimum/maximum value (0 or 255). Then it will store these new RGB pixel values into a PPMPixel array.

Lastly, the writeImage function will create a new PPM file writing both the header (Using the stored width, height, and global RGB_MAX) and the new RGB pixel data that we found using the apply_filter function. Throughout this whole process, there are error checks to make sure the input and output are correct (For example we check to make sure the PPM file has a format of P6). This entire process can be seen in the code on my github (linked above).

Concurrency:

In this project, concurrency is implemented using pthreads. We used pthreads to evenly divide up the work (unless there was an uneven amount of work, in which case the last thread would handle the extra) of applying the Laplacian filter to the inputted image. This is a great use of threads because you can divide the work of applying the filter into small pieces that do not rely on one another. I handled possible race conditions by using both join and mutex. I use join to make sure that all threads wait for each other, to make sure all threads finish their work. I used mutex to lock the result element of the parameter structure (the new/output pixel data), to prevent a thread from reading old/an outdated pixel value. This means that only one thread at a time will be able to change results(The rest of the threads will have to wait), preventing inaccurate outputs.

Experiment:

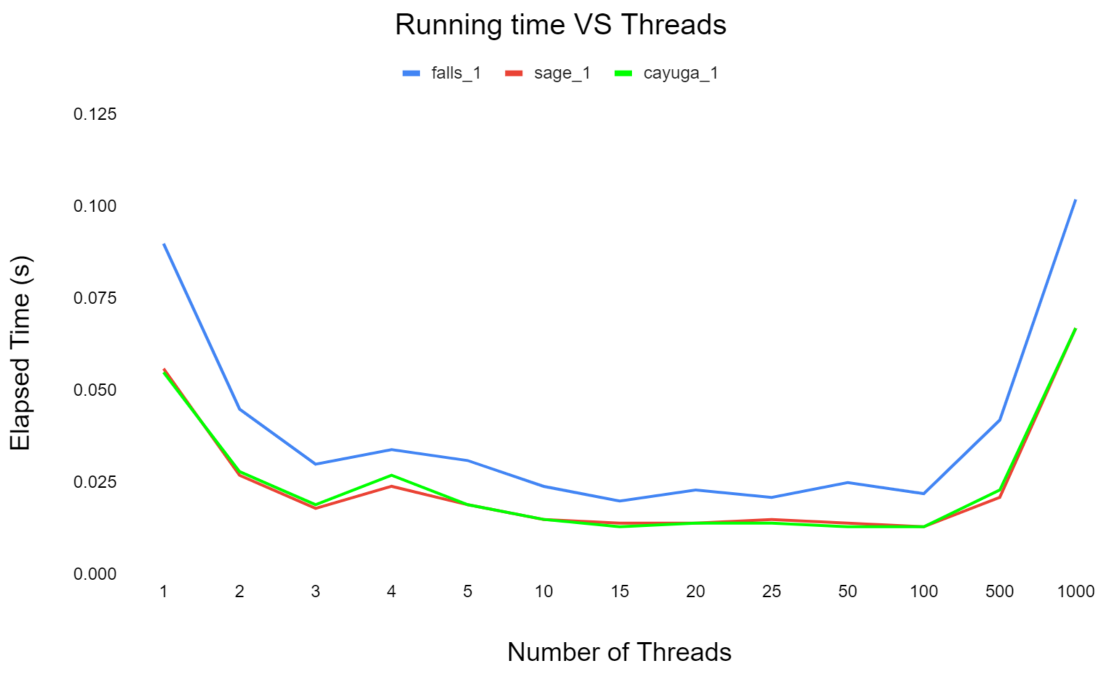
For my experiments, I found the run time to apply filters on three different images falls_1.ppm (width = 1080, height = 720), sage_1.ppm (width = 800, height = 600), and cayuga_1.ppm (width = 800, height = 600) for differing number of threads between 1 and a 1000. To control for variables I did this experiment in one session on the same lab machine (cf162-17).

Results:

Here is a table showing my results (Running time is in seconds):

Number of Threads	falls_1	sage_1	cayuga_1
1	0.09	0.056	0.055
2	0.045	0.027	0.028
3	0.03	0.018	0.019
4	0.034	0.024	0.027
5	0.031	0.019	0.019
10	0.024	0.015	0.015
15	0.02	0.014	0.013
20	0.023	0.014	0.014
25	0.021	0.015	0.014
50	0.025	0.014	0.013
100	0.022	0.013	0.013
500	0.042	0.021	0.023
1000	0.102	0.067	0.067

Here is that same data in a graph (This may be slightly deceptive because the x-values do not increase incrementally)



Link to graph

We can see from the results that the bigger image (falls_1) took longer to apply the filters, than the two smaller images. The two smaller images had the same size (800x600) and therefore had nearly identically running times, this makes sense since applying the filter on the pixel data, which will be the same size, so the only difference will be the numbers in the pixel data, which shouldn't increase the running time. The difference in the run time simply shows that bigger images take longer to process.

We can also see the dramatic difference number of threads makes. In the data we a dramatic drop applying the first several threads, then it smooths out to a relatively similar run time when we have more threads, lastly, run time peeks when we get up to a stupendous number of threads. This shows that the first several threads dramatically decrease runtime, having two threads running instead of one cuts the run time in half, and having three cuts it into roughly a third of the original run time. Then once we have enough threads (In this case 10-50) we reach the lower-bound of the runtime where we can not decrease the runtime any further using threads. The high runtime for 500 and 1000 threads is probably due to how long it is taking to creating that number of threads, and run the threadsfn function that many timeless outways the benefits of having threads (As there are so many threads, doing so little work).

Conclusion:

Overall, throughout this project, I learned a lot about concurrency, using threads, and race conditions. I learned a lot about how effective concurrency is, I was able to decrease my runtime of applying filters with threads to about 20% of the runtime without threads. I also learned more threads aren’t always just better as the runtime with 1000 threads was worse than the runtime with 0. Additionally, I learned a lot about how to implement concurrency using thread to split work on a data structure up, how to implement join to make sure all your threads finish, and how and why to use mutex to lock off the information you are manipulating. I briefly ran imath before I had either of these implemented (join and mutex) and got to see how important they were when my output was incorrect. Lastly, I also really struggled with bugs and therefore learned a lot about how to use gdb and Valgrind to detect and find bugs.